EvoCrawl: Exploring Web Application Code and State Using Evolutionary Search

by

Xiangyu Guo

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science

Department of Electrical and Computer Engineering
University of Toronto

EvoCrawl: Exploring Web Application Code and State Using Evolutionary Search

Xiangyu Guo
Master of Applied Science

Department of Electrical and Computer Engineering
University of Toronto
2023

Abstract

As the demand for web applications continues to grow, the focus has shifted towards user protection and vulnerability detection. Triggering the vulnerabilities requires 1) the vulnerable code to be executed and 2) the web application to be in the required state. While a previous scanner, AuthZee, combines evolutionary search and traditional crawling to explore application codes with complex interactions, it overlooks state transitions within the web application. To bridge this gap, we extend AuthZee to EvoCrawl, a black-box scanner that is capable of exploring application states and codes. EvoCrawl uses an evolutionary search approach to explore sequences of actions leading to different application states, and infers new states by observing the appearance of new elements or values in web application responses. In an evaluation against four state-of-art scanners, EvoCrawl achieves a 48% increase in code coverage on average and submits HTML forms 14 times more frequently than the next best tool.

Acknowledgements

I would like to express my gratitude to my supervisor, Professor David Lie, for his unwavering guidance and invaluable support throughout my academic journey. His profound insights and thoughtful feedback have consistently proven indispensable to this project. Over the past two years, I have learned a wealth of knowledge and gained numerous research skills from him, extending far beyond the field of security.

I would also extend my gratitude to my lab mates, Cassie, John, Shawn, Wei, and Eric for their invaluable feedback and engaging discussions that greatly contributed to the improvement of both my presentations and projects. I am thankful to my partner and friends for all their support and care over the past two years.

I am truly grateful for the support I have received from my parents and family. Their consistent belief in me has always been the most important source of motivation throughout my entire life.

Finally, thank you for the financial support from the University of Toronto and the Department of Electrical and Computer Engineering.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

From 2017 to 2021, there have been notable shifts in the prevalence of Cross-Site Scripting (XSS) and Broken Access Control vulnerabilities, as reported by OWASP [20, 19]. Broken Access Control has risen to become the most prevalent vulnerability, shifting from the Top 5 to the Top 1, while XSS has moved to the Top 3. These changes underscore the persistent significance of both XSS and Broken Access Control as prevalent web vulnerabilities. As a result, it is important to detect these vulnerabilities so they can be eliminated. There are two main approaches to detecting such vulnerabilities so they can be removed: static analysis and dynamic analysis. On the one hand, static analysis tools [36, 30, 17, 18] require the applications' source code, which limits their adaptability to applications written in other programming languages. On the other hand, dynamic analysis can only detect vulnerabilities if they occur during the tool's exploration of the application. As a result, for a dynamic analysis tool to be effective it must explore as much of an application's code and state as possible to try and trigger vulnerabilities. However, previous dynamic analysis tools [14, 9, 16, 7, 24, 6, 33, 2, 28, 21] only focus on exploring code, and have only simplistic (if any) heuristics for trying to explore application states. For example, they may try to use simple rules for detecting and submitting web forms to modify the web application database. As a result, even previous scanners that attempt to explore application state [14, 9, 24, 6] still fail to explore a significant portion of web application code.

To overcome these challenges, we introduce EvoCrawl, which explores more code by actively trying to explore more states, both server-side and client-side. The server-side state is stored on the web application server, typically in the database, and is modified when the user uses a web client (i.e. a browser) to submit inputs that are stored on the server. The client-side state exists on the web client and is modified when certain client-side events are triggered, which may cause other web elements to become enabled or visible on the web client. EvoCrawl navigates through the states of web applications by interacting with web elements on web pages. It employs interaction sequences to generate various requests, including AJAX requests and POST requests. Asynchronous JavaScript and XML (AJAX) requests can asynchronously update web pages and influence client-side states, while POST requests have the capability to alter the application's database, consequently changing server-side states. The design of EvoCrawl is driven by three insights:

1. application state includes both server-side and client-side state;

2. to explore client-side state efficiently, one needs to infer dependencies among interactions; and

3. state exploration should be combined with traditional crawling that collects URLs and pages for a state search.

Some previous work does not take into the client-side state at all [6], while others explore client-side events without understanding the dependencies between them [24, 9]. Without understanding the dependencies, a scanner will not explore certain sequences, as it will fail to trigger earlier events that later events depend on. EvoCrawl explores both types of server- and client-side states efficiently and uses a fitness function that monitors several features to infer which interactions trigger new client- or server-side states, together with an evolutionary search algorithm, to focus its search more intensively on interactions that lead to more codes and states.

EvoCrawl also incorporates a detector module that effectively identifies IDOR (Insecure Direct Object Reference) vulnerabilities and XSS (Cross-Site Scripting) vulnerabilities. The IDOR vulnerability detector (IVD) within EvoCrawl not only automatically categorizes resources but also exhibits a low rate of false positives when detecting IDOR vulnerabilities. Inspired by BlackWidow [9], the XSS vulnerability detector (XVD) injects XSS payloads containing unique integers into every feasible input field. By monitoring and tracking these integers, the XVD can identify the relationships between input sources and sinks, subsequently exposing the payloads.

We implement EvoCrawl based on AuthZee [14] leverage its structure of combining traditional crawl and evolutionary crawl. Furthermore, EvoCrawl extends AuthZee to delve deeper into both server-side and client-side states while expanding its capability to identify various types of vulnerabilities. Our enhancements encompass the simple crawler, evolutionary crawler, and vulnerability detection modules within AuthZee. Notable distinctions between EvoCrawl and its predecessor include:

State-aware. EvoCrawl has the capability to identify and detect changes in both client-side and server-side states.

Crawler Module. EvoCrawl's Crawler Module extends AuthZee's simple crawler by not only crawling static links but also capturing events that influence the web application's states.

Evolutionary Search Module. EvoCrawl's Evolutionary Search Module adopts AuthZee's evolutionary algorithm while modifying the fitness function to explicitly reward sequences that can change the application's states.

Vulnerability Detector. EvoCrawl's IDOR Vulnerability Detector uses a recorder-replayer structure, significantly decreasing the misclassification rates. Moreover, EvoCrawl's vulnerability detector supports the detection of XSS vulnerabilities.

We evaluate EvoCrawl against 10 modern web applications and reveal 6 zero-day IDOR and XSS vulnerabilities. We have responsibly disclosed all vulnerabilities, and the developers have either fixed or acknowledged all except two of them. In order to assess the effectiveness of the crawler module (CM) and the evolutionary search module (ESM), we compared them against four modern black-box scanners: AuthZee [14], BlackWidow [9], JAK [24], and CrawlJAX [16]. We also test the performance of the vulnerability detector module by collecting the number of vulnerabilities discovered and the number of false positives.

The followings are our main research contributions:

- We identify that successfully and efficiently exploring client-side state is a significant impediment to web application exploration, which should be overcome for web vulnerability scanners to increase code coverage and find vulnerabilities.

- We present EvoCrawl, which employs the structure of AuthZee [14] that combines the evolutionary search with standard crawling. EvoCrawl expands AuthZee's capabilities to not only reveal links but also explore and interact with the states of the web application.
- We integrate all the modules into EvoCrawl and evaluate its performance by comparing it against 4 modern scanners on 10 web applications. EvoCrawl successfully identifies seven zero-day bugs in WordPress, HotCRP, Kanboard, and ImpressCMS. It achieves an average code coverage increase of 48% and outperforms BlackWidow by submitting HTML forms with the POST method 14 times more frequently.

Chapter 2

Motivation

As described above, EvoCrawl focuses both on modifying server-side and client-side states. To modify the server-side state, a user uses the web client to submit inputs, which are then stored in the web application's persistent database. BlackWidow and Enemy of the State [9, 6] accomplish this by submitting HTML forms. However, HTML forms are not the only methods for submitting inputs. For instance, consider the example in Figure 2.1, which displays a screenshot of the post-authoring page in WordPress. Users use the page to create and modify posts and then can submit them to be posted on the application. Upon inspecting the page's HTML, we can observe buttons such as the "Publish" button that do not belong to any form elements. Instead, WordPress developers use Javascript functions, bound to the publish button, to submit the inputs. In contrast to BlackWidow [9] and Enemy of the State [6], CrawlJAX [16] submits injected inputs by clicking every button on the web page. Prior to clicking each button, CrawlJAX injects texts into every input field to ensure a successful injection after clicking. However, achieving a successful injection sometimes necessitates the scanner to interact with web elements in a specific order. For example, the crawler might click the "cancel" button before selecting the "submit" button. Additionally, certain input fields require values that scanners cannot infer. Rather than entering incorrect values, it is more effective for scanners to leave those fields unfilled. Some crawlers use heuristics to try and submit values, they will miss these specific orders. In contrast EvoCrawl's evolutionary search tries different orders of interactions and is able to successfully submit inputs in such cases.

Other crawlers, such as BlackWidow, may try different interaction orders but are inefficient because they are not aware of dependencies between interactions. EvoCrawl infers dependencies between interactions and incorporates these into the reward function its evolutionary search uses to prioritize more promising interaction sequences. In Figure 2.1, for example, clicking the "+" button labeled *Step 1* enables a list of new buttons and makes them interactable with the scanner. CrawlJAX captures these client-side changes by comparing the DOM after each interaction. BlackWidow and JAK capture these elements by overriding the `addEventListener` function and assuming that all JavaScript events can cause state transitions on the client side. However, CrawlJAX and JAK do not infer dependencies among JavaScript events, and so may not trigger events in the right order. For instance, to add a code snippet to the post publishing page, the scanner needs to first click the *Step 1* button and then click the *Step 2* button. Similarly, they may not be able to invoke the Javascript handlers for the newly added elements since the Javascript functions may only be added

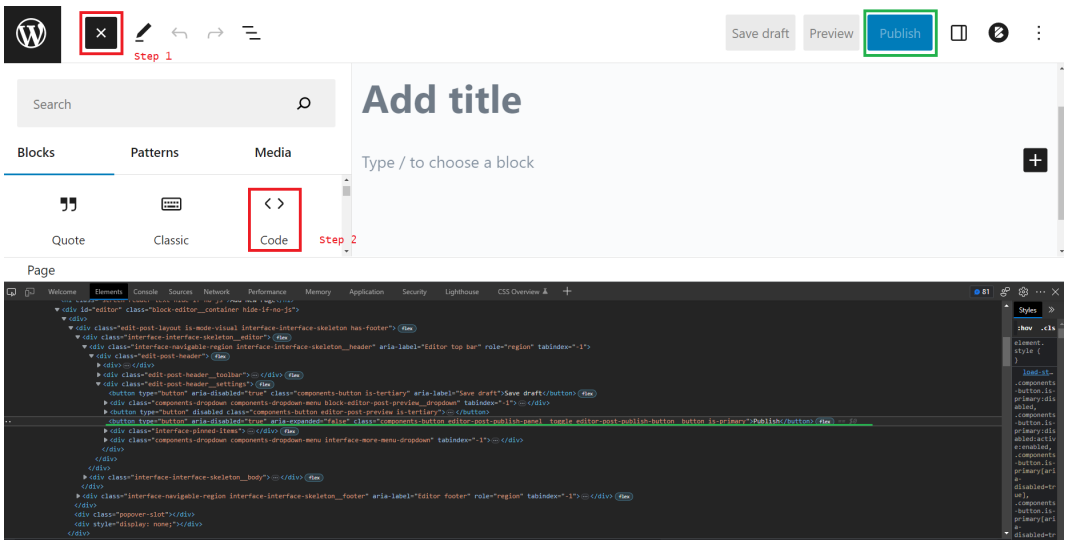

Figure 2.1: post-new.php?type=page for WordPress

to the DOM after the *Step 1* button has been invoked. To prevent overlooking such dependencies, BlackWidow attempts all possible combinations of JavaScript events to explore more client-side states. However, since not all events are related to one another, BlackWidow spends unnecessary time investigating dependencies between unrelated events, which ultimately limits what codes can be covered within a certain time period.

To tackle the aforementioned challenges, EvoCrawl incorporates an evolutionary search module that is derived from AuthZee's [14] evolutionary crawler. It considers the entire web page as the search space and extracts all interactable elements to generate interaction sequences. Additionally, during the execution of a sequence, ESM continuously monitors the DOM of the current page. If the DOM partially changes following the interaction with a web element, ESM records the newly revealed elements as dependent elements. In subsequent generations, ESM introduces mutations to the sequence by randomly selecting some of these dependent elements and inserting them after the web element that triggered the changes. For example, if an interaction sequence involves clicking on the *Step 1* button, ESM identifies the newly revealed buttons, including the *Step 2* button, as dependent elements. During the mutation stage, ESM will try different sequences of these buttons while respecting the dependencies between them. By satisfying these dependencies, the ESM is more likely to generate valid sequences of interactions.

Chapter 3

Background

In this chapter, we provide essential information to comprehend the thesis. We begin by giving a brief overview of web crawling. Subsequently, we proceed to elucidate the concepts of client-side and server-side states. Moving forward, We clarify the role of DOM events, acting as triggers for client-side state transitions within the framework of EvoCrawl. Furthermore, we introduce rrweb, a third-party tool employed to capture all events executed by the crawler, outlining its importance in our research. Lastly, we present an overview of vulnerabilities targeted by EvoCrawl: IDOR (Insecure Direct Object References) and XSS (Cross-Site Scripting).

3.1 Web Crawling

Web crawling serves the purpose of indexing web pages by systematically traversing each page and following all available links. Web crawling may also involve making edits to the pages. However, for vulnerability detection, the crawler needs to do more than just find new pages. It must navigate the applications to different states because certain vulnerabilities can only be identified when the applications are in specific states. By driving the application to these states, the crawler increases the chances of detecting such vulnerabilities.

3.2 Server-side States

Server-side states primarily pertain to the storage system of the web application, encompassing the application's data. This data incorporates various objects, such as posts, comments, uploaded files, projects, and more. Server-side states can have a significant impact on the crawler's performance since certain pages may only be accessible when the application is in a specific state. To illustrate, consider a scenario where the crawler sends a POST request and generates a new object by filling in an HTML form. In response, the application generates a page specifically for this new object. This page can only be accessed when the application is in the state where the crawler has successfully created the object.

3.3 Client-side States

Client-side state transitions largely rely on Ajax, an acronym for Asynchronous JavaScript and XML. This technology is widely employed to achieve dynamic updates within a web page's Document Object Model (DOM). It empowers web applications to execute asynchronous requests and obtain responses without necessitating a complete page reload.

For instance, developers can link a JavaScript function to an element using attributes like "onclick" or by employing the addEventListeners API. When users interact with these elements, the associated function is triggered, subsequently initiating Ajax requests to the server. The partial updates brought about by these requests can reveal new elements, further contributing to generating new requests, encompassing GET and POST requests, among others. This underscores the importance of the crawler effectively exploring client-side state transitions, which is essential to ensure comprehensive coverage of the web application.

3.4 DOM Events

DOM events are triggered when interactions with web elements lead to updates in the DOM without necessitating the loading of a new page. EvoCrawl utilizes these DOM events to initialize client-side state transitions. To activate the DOM events, EvoCrawl starts by finding elements linked to these events and then stimulates the events through interactions with these elements. To find the event-associated elements, EvoCrawl adopts a dual monitoring strategy, observing both the browser's URL field and the web page's DOM after every interaction with a web element. If the URL remains consistent but changes arise within the DOM, such as the emergence of new visible elements, EvoCrawl recognizes the relevant elements as triggers for DOM events.

3.5 rrweb

rrweb [27] (short for recording and replaying web) is a third-party tool we used to capture the events on the client side. The tool operates through two distinct modules: recording and replaying. Whenever an event occurs, the recording module captures a snapshot of the involved element along with a corresponding timestamp. Subsequently, EvoCrawl relies on these captured snapshots and timestamps to replay every interaction executed by the crawlers. Further details of the EvoCrawl's recorder and replayer are presented in Section 4.2.5.

3.6 IDOR Vulnerabilities

An Insecure Direct Object References (IDOR) vulnerability is a form of broken access control vulnerability that renders a system susceptible to unauthorized access [12]. This vulnerability arises when a web application inadequately validates and authorizes user requests, thus enabling attackers to reach resources that are normally beyond their permissible access. Notably, attackers need to craft specific URLs since the endpoints implicated in IDOR cannot be accessed through the user interface. By exploiting an IDOR vulnerability, attackers can circumvent security measures and access sensitive data.

IDOR vulnerabilities can lead to both horizontal and vertical privilege escalation. Horizontal privilege escalation refers to situations where an attacker gains access to resources or data belonging to other users at the same privilege level. Vertical privilege escalation, on the other hand, occurs when an attacker gains access to resources or data at a higher privilege level than they are authorized for. This allows them to access information reserved for privileged users or administrators.

3.7 XSS Vulnerabilities

Cross-Site Scripting (XSS) vulnerabilities pose a significant threat as they enable attackers to execute scripts in a victim's browser remotely and circumvent the same origin policy [35]. Reflected XSS vulnerabilities specifically occur when a server receives an HTTP request containing a malicious script and includes the script in the immediate response without proper sanitization. Consequently, the victim's browser executes the script upon receiving the response, leading to potential exploitation.

Stored XSS vulnerabilities occur when a web application stores malicious scripts directly into its database and subsequently includes them in responses when users request access to the affected content. This type of vulnerability poses a security risk as the stored script can be executed whenever the compromised content is loaded or displayed, affecting all users who access it.

Chapter 4

Design

EvoCrawl builds upon the work of AuthZee [14], leveraging its twin crawlers to explore web applications: the Simple Crawler (SC) and the Evolutionary Crawler (EC). AuthZee's Evolutionary Crawler (EC) is capable of executing complex interactions on each web page using a genetic algorithm. However, the EC requires more time to crawl each page compared to traditional crawlers, as the genetic algorithm needs to take multiple generations to evolve sequences that meet AuthZee's goals. Thus, AuthZee incorporates a traditional crawler that solely explores static links (*anchor* elements) within the page's HTML.

The primary goal of AuthZee is the detection of IDOR vulnerabilities. Therefore, both the simple crawler and the evolutionary crawler are mainly designed for discovering new links and capturing new requests within the target web application. However, to comprehensively explore the web application's code base, it is important for the crawler not only to visit all existing web pages but also to navigate the web application to different states. Recognizing these limitations, we have designed EvoCrawl, a black-box scanner that not only thoroughly explores the application's codebase but also effectively navigates through its diverse states. EvoCrawl shares a similar modular structure with AuthZee, comprising the crawler module (CM) for swift application traversal and the evolutionary search module (ESM) for facilitating complex interactions with the application. These modules (CM and ESM) serve as the counterparts to AuthZee's simple crawler and evolutionary crawler, respectively

However, EvoCrawl enhances the vulnerability detector of AuthZee, effectively reducing the occurrence of false positives, while also expanding its capabilities to integrate other forms of vulnerability detection. Presently, it not only facilitates IDOR detection but also includes XSS detection. Our primary objective for EvoCrawl is to prioritize the comprehensive exploration of the application

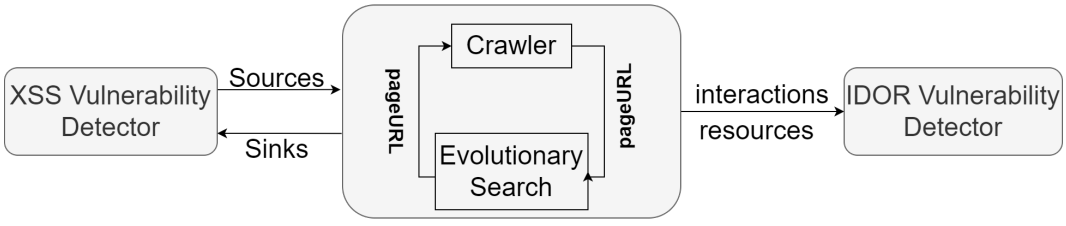

Figure 4.1: Block Diagram of EvoCrawl

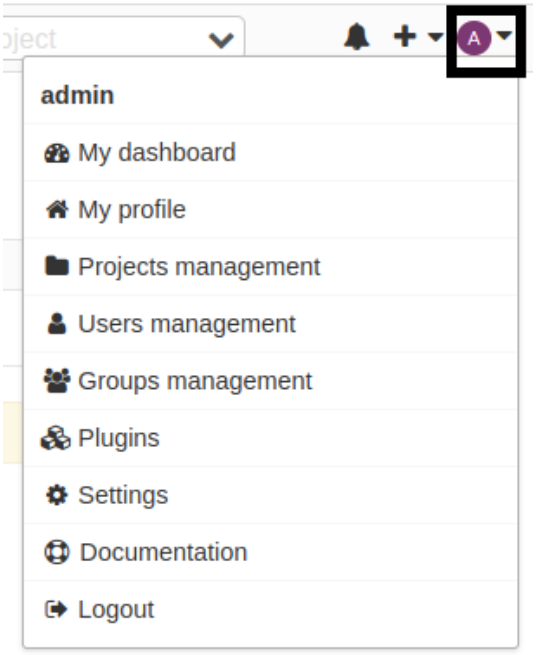

Figure 4.2: The Dynamic Page of Kanboard

itself, rather than exclusively concentrating on detecting specific types of vulnerabilities.

EvoCrawl is designed to possess the ability to change both client-side states and server-side states. As explained in chapter 3, transitions of client-side states might trigger partial loading of a component on the page. Failing to initiate these transitions could cause the crawler to overlook static links or other elements present within the component. In order to cover more sections of the application's codebase, EvoCrawl necessitates the capability to manipulate server-side states. This is crucial because certain lines of code can only be executed when the application is in a specific desired state. For instance, a user of HotCRP must first submit a paper before gaining access to pages for paper editing and review.

AJAX is a commonly used technique for modifying client-side states, and it's known that certain links and input fields may only become visible after related AJAX requests are triggered. Illustrated in Figure 4.2, clicking the button enclosed in the black box triggers the appearance of a drop-down menu, containing numerous new links. If the scanner does not successfully capture this client-side state transition, moving from a state lacking the menu to one that incorporates it, it will overlook the new links and consequently fail to crawl the associated new pages. To ensure EvoCrawl comprehensively uncovers all these dynamic elements, we have designed both the CM and ESM to be aware of DOM events. The CM stores DOM events directly as part of its seeds (further details in section 4.1), while the ESM uses triggering DOM events as a reward during the evaluation of each sequence. Furthermore, we design the evolutionary search module to be capable of inferring the dependencies among the DOM events, since some of them have to be triggered together to cause further mutations on the DOM. The ESM captures these dependencies by monitoring whether any dependent elements are revealed after each interaction. Dependent elements are elements that can only be visible after the crawler triggers the corresponding DOM event. If any of the dependent elements are able to trigger other events. The ESM can infer the dependencies between the triggered

event and the new events.

To achieve a comprehensive exploration of a web application, it is crucial to not only modify client-side states but also manipulate server-side states. This is because certain pages become accessible only when the application is driven to specific server-side states. For instance, in the case of HotCRP, a conference reviewing application, its paper-editing page and paper-reviewing page can only be accessed after a paper has been submitted. To address this requirement, both EvoCrawl's CM and ESM have been purposefully designed to be capable of changing server-side states. This is accomplished by inserting data into the application's storage system, such as databases. The CM plays a role in this process by systematically filling out every HTML form it encounters on each web page. By doing so, it provides a mechanism for inserting data into the storage system, thereby altering the server-side state. On the other hand, the ESM relies on the detection of successful input insertions as a reward for the fitness function.

Figure 4.1 illustrates the Block Diagram for EvoCrawl. The Crawler Module plays an important role in swiftly crawling through the web application and collecting URLs, which are subsequently passed to the Evolutionary Search Module for exhaustive exploration. Simultaneously, if the Evolutionary Search Module uncovers previously undiscovered links, it shares them with the Crawler Module, enabling further crawling. Throughout the scanning process, both modules exchange URLs to the IDOR vulnerability detector (IVD). The IVD classifies the URLs and assesses the access control level of the private ones. Additionally, the XSS vulnerability detector generates payloads for both the Crawler Module and the Evolutionary Search Module, while monitoring the successful execution of these payloads.

In summary, the main differentiating features of EvoCrawl over AuthZee.

- CM vs. SC: Both SC and CM crawl *anchor* elements, while CM additionally crawls DOM events and HTML *form* elements. Because of solely targeting anchor elements, SC misses dynamically updated links on the web page. By not focusing on forms, it limits its ability to detect only IDOR vulnerabilities. Moreover, it lacks the capability to modify server-side states. By crawling DOM events, CM gains the ability to trigger transitions on the client-side state, enabling it to observe and capture the new elements added to the web page. This allows CM to explore the dynamic behavior of the web application while crawling HTML forms enables it to insert different inputs into the web application and therefore change server-side states.

- ESM vs. EC: Both ESM and EC use evolutionary search to guide the crawling, but ESM uses another fitness function with different reward variables. While EC targets all interactable elements on the web page, it does not explicitly reward sequences that can modify the web application's states. Furthermore, it does not recognize that clicking on static links and input insertion should not occur in the same sequence. This oversight is problematic because clicking on static links can lead the EC to another page, causing a loss of client-side states such as triggered DOM events and texts typed by the EC. On the other hand, ESM's fitness function heavily rewards sequences that trigger DOM events and insert inputs while tracking the dependencies among these DOM events. This consideration is crucial as certain events must be triggered simultaneously to update the DOM of the web pages. Moreover, ESM dynamically memorizes the *anchor* elements it clicks on and removes them from sequences in subsequent generations. As a result, after several generations of evolution, ESM can efficiently discover sequences that exclusively focus on modifying the states of the web application.

- Vulnerability Detectors: The VDs of two scanners can detect IDOR vulnerabilities. However, EvoCrawl's VD uses different public filters and supports XSS detection. AuthZee's public filter utilizes two different users to crawl the application simultaneously. URLs visited by both users are classified as public. However, due to the random nature of the evolutionary search, it is highly probable that the two users explore different areas of the application. As a result, the second user may miss URLs that the first user visited, leading them to be misclassified as private ones. To fix these, EvoCrawl's IDOR vulnerability detector adopts a recorder-replayer structure, where the replayer replays each interaction in lockstep. This approach reduces the risk of missing URLs. Moreover, EvoCrawl's vulnerability detection extends beyond IDOR to include XSS vulnerabilities, and it can easily integrate other types of vulnerabilities in the future.

4.1 Crawler

While AuthZee's simple crawler only concentrates on crawling *anchor* elements (static links), it lacks awareness of DOM events and form elements. Therefore, it is unable to modify both client-side and server-side states, limiting its exploration of the application codes that can only be executed when the application is in the desired state. On the other hand, EvoCrawl's crawler module places emphasis on these elements and is capable of finding new links as well as new states.

The crawler module (CM) manages the seeds by storing them in a queue. Each seed consists of a URL and an associated event represented as a $< URL, EVENT >$ pair. The EVENT value can be either empty or a CSS selector of a specific web element that triggers the corresponding DOM event. During each iteration, the crawler follows a three-stage crawling process, which includes links crawling, events crawling, and forms crawling.

During the links crawling stage, the CM performs the following steps. First, it executes the seed by navigating to the URL specified in the current seed. If the value of EVENT is not empty, the associated DOM event is triggered. Once the seed execution is successful, the crawler extracts all `href` values from the anchor elements on the web page. These `href` values are then used to construct new seeds, which will be added to the end of the queue if they are not already in the queue or visited before.

During the events crawling stage, the CM interacts with each interactable element on the web page to determine if it can invoke a DOM event. If, after each interaction, the DOM changes without refreshing the page or navigating to other pages, the crawler identifies the corresponding element as the trigger for a DOM event and generates a new seed. This seed includes the URL of the current page and the CSS selector of the element that triggered the event.

The CM does not combine different events to construct new seeds. If there are two events on the web page: Event 1 and Event 2, the crawler will only generate two new seeds $< URL, Event1 >$ and $< URL, Event2 >$ but not $< URL, Event1, Event2 >$ since it cannot track the dependencies among these events. It avoids blindly combining them in case the CM wastes time crawling on the same DOM.

In the form crawling stage, the CM first collects all the forms by identifying the elements with the `form` tag and then tries to submit all of them immediately. For each form, it tries to interact with all the elements inside it sequentially. During the submission, If the CM detects any mutations on

the DOM of the form, it will capture and interact with the new elements dynamically. For example, after clicking on the submit button, if a confirmation window pops up, the CM can detect the new elements inside the window and also interact with them.

4.2 Evolutionary Search

The design of the ESM is inspired by the evolutionary crawler of AuthZee [14] but with a different fitness function. Unlike the traditional approach of crawling, AuthZee's evolutionary crawler carries out sequences of interactions on the web page. For each seed (page URL), it first browses to the page URL and extracts all interactable elements from the page followed by constructing them into different genes, which is defined as an element-interaction pair. For example, an `input` HTML element can have genes such as `input-click` or `input-typeText`. Then, it employs the genetic algorithm to evolve sequences that explore the states of the current page. Its termination does not depend on whether the evolutionary search has reached an optimal point. Instead, AuthZee set a maximum number of generations that the EC spends on each page. Then in each generation, the evolutionary crawler produces gene sequences, mutates gene sequences, evaluates gene sequences, and then selects sequences to pass on to the next generation.

EvoCrawl follows a similar workflow as AuthZee but introduces a different goal for the evolutionary search. While AuthZee primarily focuses on discovering new links, EvoCrawl's ESM has the objectives of revealing new links and injecting inputs. However, achieving both tasks simultaneously presents challenges. Normally, generating a new object may be hindered if the ESM clicks on static links (anchor elements on the web page) since they navigate the crawler to new pages and cause a loss of the client-side state required for object generation. Moreover, the search space for the evolutionary crawler is the entire web page but not one single form. Due to the number of static links on one page, they can take up a large portion of the total genes, which makes it hard for the ESM module to find sequences that can successfully generate objects. Therefore, during the execution of each sequence, the ESM keeps memorizing the static links it clicks on and dynamically removes those genes after sending the links to the seed queue.

Additionally, the ESM utilizes a linear fitness function that offers rewards for successful state transitions. Further elaboration on the fitness functions can be found in subsection 4.2.3. To enhance the efficiency of the ESM, it incorporates an early termination mechanism if the ESM fails to find any sequences with fitness scores exceeding the specified threshold. By employing a linear fitness function, EvoCrawl simplifies the evaluation process, making it easier to tune and optimize the exploration of sequences.

4.2.1 Producing Sequences

Then, to produce new sequences, ESM either randomly combines these genes or crossover previous sequences. For crossovers, it uses sequences selected from the previous iterations and recombines them to be new sequences. For example, if Sequence 1 and Sequence 2 are two sequences with the highest score, the ESM will concatenate the first half of Sequence 1 with the last half of Sequence 2 to produce a new sequence. Also, we enforce one of the submit buttons at the end of each sequence to increase the chances of successfully submitting the inserted inputs.

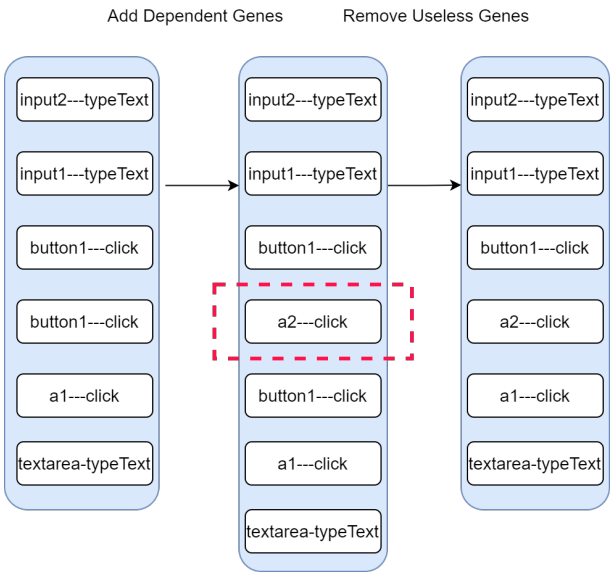

Figure 4.3: Mutating Process of an Example Sequence

4.2.2 Mutating Sequences

After producing new sequences, the mutation function will parse and mutate each sequence by randomly adding dependent genes. Dependent elements are elements that can only be visible after the crawler interacts with the element that triggers the DOM event. For example, if there is one anchor element called a2 and one button element called button2 and a2 will be visible after interaction with button1, the mutation process of a sequence is presented in Figure 4.3. Since a2 is dependent on button1, it has been added after button1 in the example sequence.

4.2.3 Evaluating and Selecting Sequences

To evaluate a sequence, the ESM navigates to the page URL and then iterates through all the genes in a sequence. In each iteration, it interacts with the corresponding element based on the interaction type. For example, to evaluate the sequence in Figure 4.3, it first types texts into input2 and input1, clicks on button1, a2, and a1, and finally types texts to the textarea. After each interaction, it checks the URL field of the browser to see whether the ESM has been navigated to another page. If clicking on an element navigates the crawler to another page, the ESM will automatically record the new URL and also send it to the crawler module, navigate back to the current URL, and continue executing the next gene. By doing this, we can limit the search space of the evolutionary algorithm to the current page and thus explore more states of it.

Based on AuthZee, ESM uses a fitness function to guide the evolution, but toward a population of sequences that can drive the web application to new states. We believe that a new state can potentially lead the ESM to find new links or successful input insertions. Initially, the ESM assigns each sequence with the same fitness score which is updated dynamically during the execution of the sequence.

The fitness function of ESM is computed as:

$$fscore = r_1 * F + r_2 * I + r_3 * U + r_4 * D + p_1 * V + p_2 * O + p_3 * T \tag{4.1}$$

where r_n is the reward parameter, while p_n represents the punishing parameter. F corresponds to the number of form submissions, I denotes the number of filled inputs, U represents the number of uploaded files, D corresponds to the number of triggered DOM events, V denotes the number of invisible elements, O represents the number of out-of-domain requests, and T corresponds to the number of failed interactions. The design of the fitness function is linear, making it easier to manipulate, and its parameters are also more straightforward to tune. We conduct a parameter search to observe the influence of each parameter, and further details are available in section 6.2.2.

While the fitness function of AuthZee's evolutionary crawler is computed as:

$$fscore = r_1^{'R} * r_2^{'I} * r_3^{'U} * r_4^{'E} * p_1^{'V} * p_2^{'O} * p_3^{'T} * p_4^{'S} \tag{4.2}$$

where r'_n and p'_n are still reward and punishing hyper-parameters. I, U, V, O, and T have the same meaning as in the fitness function of EvoCrawl. R denotes the number of requests with unseen target URLs. E represents the number of newly visible elements after interactions with the web page. Finally, S corresponds to the number of requests targeting visited URLs.

For EvoCrawl's ESM, the fitness score determines how promising the sequence is in exploring the states of the current page. On the other hand, AuthZee's evolutionary crawler tends to find sequences that can send requests with unseen URLs. Similar to AuthZee, EvoCrawl rewards the sequence if the ESM successfully types texts to an input field and uploads a sample file. If, during the execution of a sequence, the crawler tries but fails to interact with an element, generates requests to an out-of-domain URL, or interacts with an invisible element, the fitness score will be decreased.

However, unlike AuthZee, EvoCrawl does not provide rewards for successfully sending requests to unseen URLs. It exempts this reward since the same sequence or the children of the sequence cannot necessarily lead to new requests in the later generations. And previous unseen URLs will become visited URLs after the execution of the sequence. Furthermore, EvoCrawl also does not punish sequences that send duplicate requests. First of all, ESM actively memorizes the page links it visits and removes them from the gene space. Therefore, it is unlikely for the ESM to send duplicate "GET" method requests to the same endpoint. Second, it is actually a desirable action for the ESM to send multiple "POST" method requests to the same URL. Although they target the same URL, these requests can have different request bodies and should not be punished by the fitness function.

EvoCrawl additionally rewards sequences when they manage to insert inputs into the storage system, which is detected by observing the web page HTML after the execution of the entire sequence. Since the generated texts for all input fields are different from each other, the ESM can check whether these texts are reflected somewhere inside the HTML. And the reflection of the texts is the sign of a successful input insertion. Moreover, the triggered DOM events are rewarded because of their ability to change client-side states. Over generations, the evolutionary crawler can produce sequences that are more likely to drive the current page to different states.

During the evaluation, we also dynamically remove genes that are not useful for the genetic algorithm. The useless genes include those that contain links visited by the crawler before or

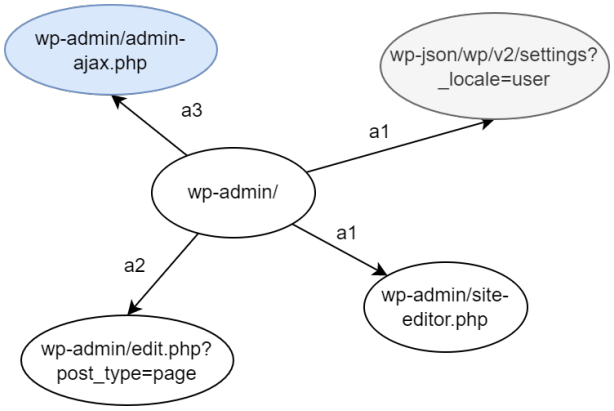

Figure 4.4: An Example Sitemap for WordPress - Blank Node: Page URL, Grey Node: Restful API, Blue Node: Ajax URL

pointing at out-of-domain pages as well as genes that can force the crawler to log out. We do this to shrink the search space of the genetic algorithm and further decrease the number of generations needed for ideal sequences.

Finally, after evaluation, the evolutionary crawler selects some number sequences with the highest fitness score to mutate to produce the next generation of sequences. We currently fix this number to be half the population size.

4.2.4 Vulnerability Detectors

EvoCrawl is designed to allow the integration of vulnerability detectors, which are activated as EvoCrawl explores the application. In this study, we demonstrate EvoCrawl's ability to detect authorization and injection vulnerabilities, as they have become increasingly prevalent [5]. In particular, we address Insecure Direct Object References (IDORs), which allow unauthorized access to private resources through specially crafted URLs. IDOR vulnerabilities are a significant concern, with more than 200 instances reported on HackerOne each month [4]. We also demonstrate EvoCrawl's ability to detect Cross-Site Scripting (XSS), a vulnerability that enables attackers to execute custom scripts on the victim's side. XSS attacks can lead to session hijacking and complete account compromise. XSS remains one of the most common web application vulnerabilities [**owasp**]. These two detectors operate independently and can be executed either individually or in conjunction with each other.

4.2.5 IDOR Vulnerability Detector

To detect an IDOR vulnerability, a scanner needs first to find the resources, filter out public resources, and then test the access control level of private ones. We avoid testing the access control level of the public resources because, unlike private ones, they are designed to be common for different users.

AuthZee's public filter uses two users at different privilege levels to crawl the application at the same time. URLs accessible by both users are categorized as public, while those accessible to only one user are deemed private. However, due to the non-deterministic nature of the evolutionary search, the crawling process involves randomization. As a consequence, the two users may explore different parts of the web application during the crawl. Consequently, certain public pages may only

be visited by the first user, while the second user may miss them entirely. This situation causes the public filter to fail in accurately classifying these pages, as they have not been seen by both users. To overcome the shortcomings of AuthZee's VD, EvoCrawl's IVD employs a recorder-replayer structure where the replayer replays every interaction of the recorder.

Now, we provide the details of how the IVD collects private resources and filters out public ones. During the crawling, the IVD automatically captures all requests sent from the browser, then extracts the request URLs and builds them into a sitemap. The sitemap indicates the dependencies among all the captured URLs. Each node in the sitemap represents a URL and the edge between the two nodes is the element that triggers the transition from the source URL to the destination URL. Figure 4.4 is an example sitemap for WordPress.

Then, the IVD uses another user (the replayer) with a different privilege level from the crawler user to test if each edge in the sitemap is accessible and also marked the corresponding edge by replaying every interaction of both the crawler module and evolutionary search module. If the replayer manages to replay the interaction, the corresponding edge will be labeled as accessible (i.e. public) to the replayer and vice versa. It is important for the replayer to correctly replay the interaction and mark the right edge since mistakenly labeling an edge can cause the IVD to misclassify an object, which leads to both false positives and false negatives. We also note that it is important that both the crawler and replayer operate on the same web application instance and in lockstep. This is because we need the replayer to see the same web application state as the recorder, to avoid missing public resources. For example, the crawler could create a public object and then subsequently delete it. If the replayer tries to access the same object on another instance, or outside of lockstep with the crawler, it might mistakenly classify the object to be private, and this will lead to a false positive when it finds it is able to access it later during detection.

After the replayer tests all the edges, we can know which resources are private by parsing the sitemap. Considering the example in Figure 4.4, if a2 element can be accessed by the replayer but a1 element cannot be accessed by it, the `edit.php?post_type=page` is considered a public resource while the `site-editor.php` is considered private. There might be different paths to reach the same resource. And If one of the paths can be accessed by the replayer, the destination node will be considered public. After gathering all the private URLs, the IVD will directly send forged requests to them and check their access control levels by analyzing received responses.

We integrate recorders into both the CM and the ESM. It automatically records all interactions from both modules and sends them to the replayer. For each interaction, the recorder generates a CSS selector for the interacted web element and the corresponding action type. Then, the replayer tries to find and interacts with this element by using the CSS selector. Figure 4.5 is an example of the crawler module and its replayer. We design the replayer to be one step behind the crawler. Before the crawler interacts with an HTML element, it will wait for the replayer to check the accessibility of that element. And after the replayer finishes, the crawler then interacts with it and moves on to the next element. The one step behind replaying can keep the replayers and the crawlers almost in the same application state.

Following the collection of private resources, the IVD tests their accessibility to unprivileged users. This evaluation involves both the crawler users and the attacker users sending crafted requests to the endpoints of private resources. Subsequently, the received responses are parsed to ascertain if they disclose data to attackers. It's important to note that the parsing process of EvoCrawl's IVD differs

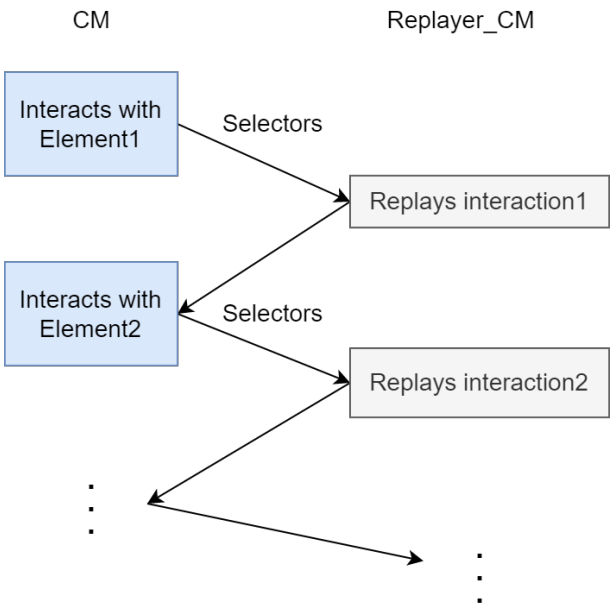

Figure 4.5: An Example Execution Trace of Crawler Module and Replayer. Arrows imply the execution order.

from AuthZee's VD. While AuthZee's approach solely involves response comparison, EvoCrawl's IVD incorporates a two-step parsing process. The first step uses keyword matching while the second step compares responses. Since some access-denied sentences are shared among most of the access-denied pages for each application, we use these sentences to identify the responses with proper access control. Based on our experiments, 5 sentences are enough for most of the tested applications. We call these sentences access-denied sentences. if the responses received by the attackers do not contain any access-denied sentences, they are further passed to the second step parser to decide whether there are broken access controls. For now, we manually collected the access-denied sentences for each application. It is possible that we fail to capture all the denied sentences. In this case, we rely on the second-step parser to check the access controls.

Similar to AuthZee's VD, the second-step parser calculates the similarity among the responses. This parser computes the similarity between responses. If the ratio of similarity between the response of the crawler and that of the attacker exceeds a predefined threshold, the corresponding private resource is deemed vulnerable. However, finding a universal threshold that is suitable for all applications is challenging. Consequently, the IVD mainly relies on the keyword-matching technique. The second-step parser is used specifically for responses that lack any access-denied sentences.

4.2.6 XSS Vulnerability Detector

We integrate the XSS Vulnerability Detector (XVD) directly into both ESM and CM. For CM, after each submission of a form, the XVD follows with an attack by replacing the input value with its XSS payload. For ESM, the XVD directly replaces all texts generated by the evolutionary crawler with the XSS payload.

The XSS payload of our XVD is similar to the payload used by BlackWidow. if the payload

injected into each input field is successfully executed by the JavaScript Engine of the browser, a unique integer will be pushed into a global list that is pre-inserted into the header of the page. The unique integer is generated according to the UNIX timestamp to avoid two input fields being inserted with the same value. And by checking the values in the global list, we can know which payload has been executed and further trace it down to the source input field.

Chapter 5

Implementation

The Evolutionary Search Module is built on top of the evolutionary crawler of AuthZee [14], while the CM is implemented using TestCafe [31]. TestCafe is an end-to-end testing framework designed for web applications and built to run on Node.js. It facilitates various user interactions with web pages, such as clicking, typing texts, and uploading files, among other functionalities.

EvoCrawl utilizes the rrweb tool, which stands for recording and replaying web, to capture client-side events. rrweb consists of two modules: recording and replaying. The recording module assigns unique rrweb-IDs with timestamps to DOM elements for event tracking. The replaying module replays interactions based on the recorded rrweb-IDs and timestamps. However, when EvoCrawl's public filter module replays interactions as another user, the rrweb-IDs may lead to incorrect elements due to dynamic DOM changes. To address this, we only use rrweb's recording module and implement our own replaying mechanism. rrweb is injected into the web page's header to capture DOM mutations and changes, including newly visible elements in the UI.

5.1 Additional Requirements

ESM and CM operate as separate processes with distinct cookie sessions. Applications like Opencart or phpBB include tokens in their page URLs that need to be matched with the corresponding cookie values. When exchanging seeds (page URLs) between ESM and CM, the token values must be automatically replaced with the appropriate ones for each module. To achieve this, both modules perform two tasks: identifying token names and replacing their values. We compare the redirect URLs after logging in from both modules to determine token names. By comparing the query strings of the URLs, we infer which parameters differ and consider them as tokens. Once the token names are obtained, the modules extract their values from the URLs and replace tokens in incoming seeds. However, this approach only works for persistent tokens present in all page URLs. Tokens that appear in only some URLs cannot be identified and captured by EvoCrawl, leading to the exchange of invalid seeds. While this slows down EvoCrawl, it does not break the entire system.

For the ESM, achieving effective results with the genetic algorithm entails tuning several essential parameters. These parameters include the sequence length, population size, number of generations or iterations, and rewards and penalties for the fitness function.

Chapter 6

Evaluation

In this section, we present the evaluation of EvoCrawl and the results from our experiments. To demonstrate the crawling ability of EvoCrawl, we compare the coverage between EvoCrawl and four state-of-arts scanners: AuthZee, BlackWidow, JAK, and CrawlJAX. Additionally, we conduct separate tests to evaluate the performance of the CM and the ESM. We assess all 6 scanners on 10 modern web applications and used the lines of code executed as the metric to measure the performance. We also conducted an analysis to understand the impact of each parameter within the Evolutionary Search Module's (ESM) fitness function. For this purpose, we run EvoCrawl with varying parameter configurations and computed the total lines of code executed across all test applications. Furthermore, we collect the number of successful HTML form submissions and how many inputs have been successfully injected into the server-side database for EvoCrawl, AuthZee, and BlackWidow. To evaluate the performance of the vulnerability detectors, we collect the number of false positives and real vulnerabilities. The experiment setup is in subsection 6.1.

6.1 Experiment Setup

Each crawler runs on a 4-CPU virtual machine with 6GB memory. The CPU type of the virtual machine is Intel(R) Xeon(R) Gold 6336Y. To ensure a fair comparison, we reset all tested web application instances prior to each crawling session. This ensures that all scanners start from the same initial state, minimizing any potential disparities caused by differing application states.

6.1.1 Configuration

We manually set up the login credentials for all the tested crawlers and prevent them from crawling on the user page and basic configuration page of each web application, because crawling on those pages may change the login credentials or cause the web application to crash. Furthermore, we preclude the crawlers from navigating to extension or plugin installation pages, recognizing the potential for compatibility issues among different plugins that could compromise the application's stability. We also manually prevent all crawlers from interacting with log-out buttons to make sure they always stay logged in. Each testing process was run for 24 hours.

For EvoCrawl's ESM and AuthZee's evolutionary crawler, all the parameters including the sequence length, the number of generations, etc. are fixed for all the benchmarks. We use the default

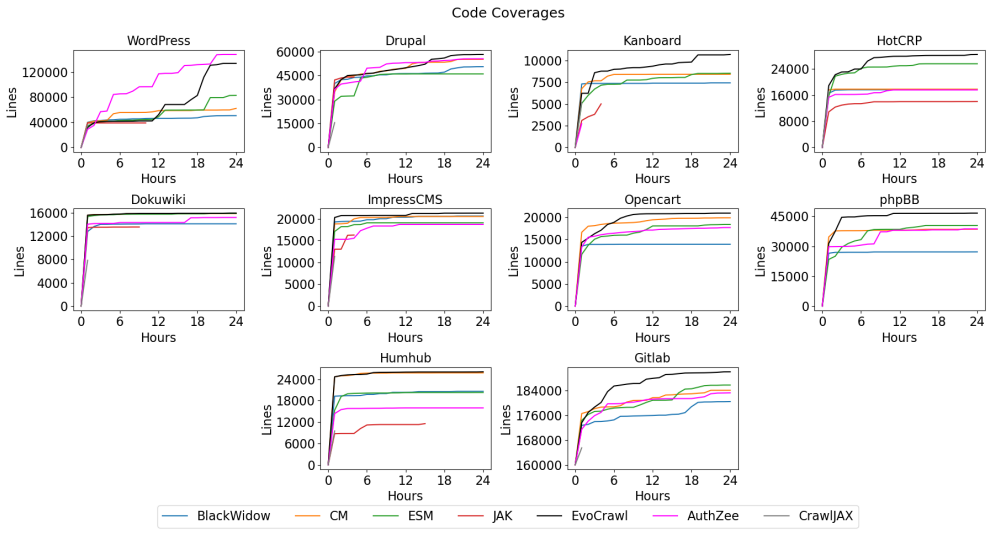

Figure 6.1: Code Coverage of Each Web Application

WordPress 371831	Drupal 790400	Kanboard 94879	HotCRP 77856	Dokuwiki 176594
ImpressCMS 212244	Opencart 118654	phpBB 251094	Humhub 534700	Gitlab 305410

Table 6.1: Total Lines of Codes within Each Web Application

settings for CrawlJAX with unlimited crawling depth and unlimited states. We also enable it to click on event handlers. For JAK, we follow the same configuration the developers provided in the example file. For the form submission experiment, since we need to search the injected texts to detect whether the form has been successfully submitted, we need the scanners to generate a unique text for each field. While EvoCrawl already supports this, we still need to modify the implementation of BlackWidow and AuthZee, since they generate the same texts for all the fields. The modification is only applied for the form submission experiment but not the coverage experiment.

6.1.2 Web applications

We select multiple types of applications to test the scanners on handling different logic, including eCommerce (Opencart-4.0.0), Blog (WordPress-6.2.2), Content Manage System (Drupal-9.3.15, HotCRP-v3.0b3, Dokuwiki-2022-07-31 “Igor”, ImpressCMS-1.4.4), Forum(phpBB-3.3.8), DevSecOps Platform (Gitlab-11.5.1), Project Manage Software (Kanboard-1.2.22), and Social Software Platform (Humhub-1.12.1). All applications are state-of-art and in common use at the time of writing.

6.2 Code Coverage

For the coverage experiments, since EvoCrawl and AuthZee use both the traditional crawl and the evolutionary crawl to interact with the web application, we also run two processes in parallel for

other crawlers to let them have the same CPU resources as EvoCrawl. We use lines of code as a metric for the coverage and generated a coverage report indicating which lines had been hit for each request sent to the server. The web application server automatically assigns a timestamp for each coverage report so that we can detect how the coverage changes along with the time. The coverage report for the PHP application is generated by using the Xdebug [34] and php-code-coverage [25]. For the application in Rails production, we use the Coverband [3] to collect the coverage result. Also, we disable the vulnerability detectors for all the scanners since we want to focus on testing the ability to crawl the web application.

Figure 6.1 presents how each tested application's coverage changes over time and Table 6.1 provides an overview of the total lines of codes within each application. phpBB necessitates an additional login procedure for accessing the administrative board. Given that jAK and CrawlJAX support only one-time login, their capability to crawl on phpBB is compromised. Additionally, in the case of Opencart, these scanners fail to extract the redirect URL from login responses. Consequently, jAK and CrawlJAX are excluded from testing on both phpBB and Opencart due to these limitations. In some cases, certain scanners finish their crawling processes before reaching the 24-hour limit, resulting in truncated lines in the graph.

EvoCrawl consistently demonstrates the highest coverage across most tested applications, outperforming other scanners. However, it's worth noting that AuthZee particularly excels in the WordPress case. Further information can be found in . In various applications, EvoCrawl surpasses AuthZee's performance by a margin ranging from 4.9% to 292.7%. Even in comparison to the most recent traditional crawler, BlackWidow, EvoCrawl still significantly outperforms it and has an improvement from 15% to 163% across different applications. Furthermore, for most applications tested by BlackWidow, the coverage only slightly increases after an hour of crawling since it spends time exploring the combinations of unrelated events. We provide a case study of why EvoCrawl achieves better coverage on certain applications than AuthZee and BlackWidow in 6.2.1.

For most applications, CM achieves similar or better results than ESM, which aligns with our expectations since ESM is slower than other scanners. ESM has to explore multiple states on each page and, therefore, handles fewer pages than CM, which is also the reason why they need to be paired together. The CM can crawl more pages in a limited period and explore more codes of the application. Still, The combination of ESM and CM largely outperforms each one of them on coverage because ESM's ability to explore various application states enables CM to explore more application codes. For instance, the ESM can install different themes on WordPress, and then the CM can quickly explore the codes of these themes. Moreover, ESM can submit papers on HotCRP and the CM can execute the application codes related to handling the paper, etc.

6.2.1 Case Studies of the Coverage Results

WordPress. AuthZee achieves better coverage over EvoCrawl on WordPress by a margin of 10.6 percent. AuthZee prioritizes link discovery, particularly evident in scenarios like WordPress, where theme installations are triggered by clicking on links. This approach enables AuthZee to activate more themes successfully, resulting in the execution of a greater volume of code.

Both EvoCrawl and AuthZee outperform BlackWidow, which mainly comes from two factors. First, BlackWidow fails to install themes because the links for the installation contain CSRF tokens that can easily expire after a short period. Since BlackWidow follows a BFS manner to crawl

the application, these links have already expired when it tries to crawl them. Second, although BlackWidow can create draft posts on WordPress, it fails to publish them. This is because the publishing post page (post-new.php or post.php) does not contain the HTML form element. Since BlackWidow only targets the form elements for generating post requests, it fails to publish the posts. However, the ESM of EvoCrawl and AuthZee's evolutionary crawler regards the entire web page as the search space, it not only can draft the posts but also manage to publish them.

HotCRP. EvoCrawl significantly outperforms both AuthZee and BlackWidow on HotCRP. This advantage is primarily due to EvoCrawl's ability to successfully submit specific forms, while the other two scanners fail to submit them. AuthZee's limitation stems from its primary focus on link discovery. As discussed in section 4.2, AuthZee's evolutionary crawler navigating through static links leads to page transitions and the loss of the original client-side state. Consequently, AuthZee faces challenges in submitting forms that require a sequence of actions, as transitioning to a new page disrupts these sequences. Furthermore, some forms trigger confirmation windows upon submission, which AuthZee fails to detect.

BlackWidow achieves lower coverage on HotCRP because it hits the "cancel" button before reaching the "save" button during the submission of these forms, while the ESM of EvoCrawl can find the sequence of interactions that omits the cancel button but click on the "save" one. This is especially important for HotCRP, as the scanner must submit certain forms before it can reach certain related code blocks. For example, it needs to first successfully submit a paper, before it can successfully crawl on the "reviews for the paper" page. Moreover, BlackWidow fails to enter the correct values for some input fields inside the form. For these forms, the ESM is also able to find the sequence that leaves them blank but only types texts to the fields that can be inferred by it.

Kanboard. AuthZee achieves the worst coverage on Kanboard, because Kanboard heavily relies on AJAX to update the DOM and AuthZee fails to detect these events. Also, EvoCrawl achieves better results over BlackWidow since it successfully creates tasks inside the projects on Kanboard. Similar to cases on HotCRP, there are input fields inside the task creation form whose values cannot be resolved by all the scanners. Instead of filling in the wrong values like other scanners do, the ESM of EvoCrawl successfully finds sequences that leave them blank and manages to create the tasks.

Opencart. The "save" button for every HTML form on Opencart is located outside the form element. Therefore, BlackWidow fails to submit all of them since it only interacts with the elements inside the form for submitting it. Nevertheless, the ESM regards the entire page as the search space for the genetic algorithm, it not only interacts with the elements inside the form but all the interactable elements on the web page, which is why it can manage to submit the forms on Opencart. On the other hand, the fitness function of AuthZee's evolutionary crawler heavily rewards clicking on new links. Consequently, it hinders the crawler to find sequences that submit forms.

For other applications, the suboptimal outcomes of AuthZee results from two main factors. First, AuthZee's simple crawler can only discover links whereas EvoCrawl's Crawler Module (CM) possesses the added capabilities of capturing DOM events and submitting forms. Furthermore, certain forms and DOM events are only detected by the CM since ESM has a lower speed and fails to find them within the 24-hour limit. This further contributes to the overall coverage of EvoCrawl. Second, AuthZee's overemphasis on new links leads to an imbalance in its fitness function, as it heavily prioritizes sequences that discover new pages. Unfortunately, this approach can result in AuthZee overlooking sequences that are instrumental in modifying the states of the applications. Notably,

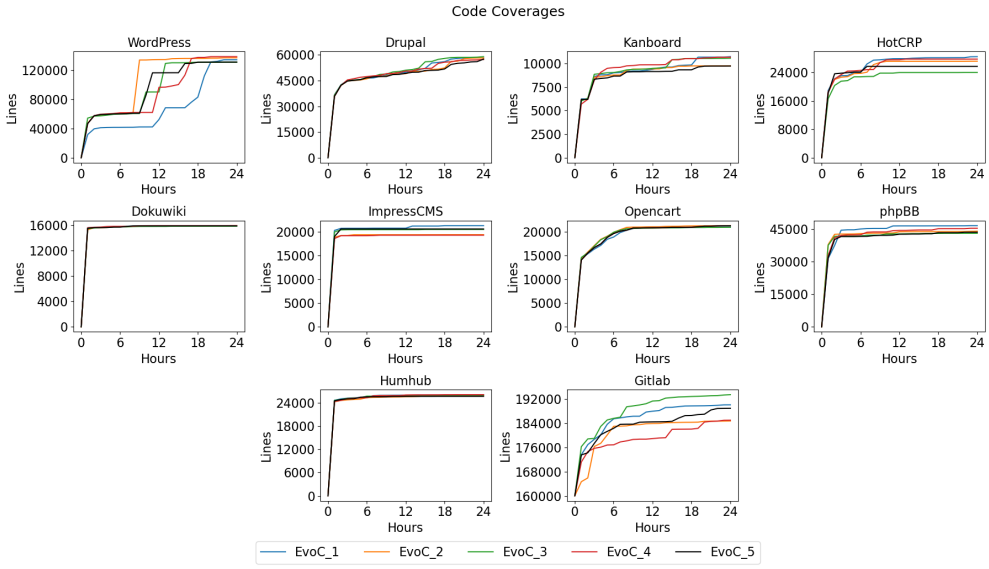

Figure 6.2: Code Coverage Results of Different Parameters

in certain cases, specific pages within these applications can only be found, or particular sections of code can only be executed when AuthZee's evolutionary crawler successfully identifies sequences that steer the application into the desired states.

BlackWidow has worse results because of two reasons. First, it spends time exploring the combination of unrelated events. As a result, it may not efficiently navigate through the application and interact with as many pages or forms as EvoCrawl does. Second, for certain forms, it fails to find the right sequences to submit them. The two factors together lead to a lower coverage achieved by BlackWidow in terms of the overall exploration of the application.

6.2.2 Parameter Search

In parameter search experiments, we modify one rewarding parameter at a time, conducting multiple experiments to understand the influence of these parameters. The following are the fitness functions tested in the experiments:

$$f_1 = 40 * F + 20 * I + 20 * U + 15 * D - 5 * V - 15 * O - 5 * T \tag{6.1}$$

$$f_2 = 40 * F + 20 * I + 20 * U + \mathbf{25} * D - 5 * V - 15 * O - 5 * T \tag{6.2}$$

$$f_3 = 40 * F + \mathbf{30} * I + 20 * U + 15 * D - 5 * V - 15 * O - 5 * T \tag{6.3}$$

$$f_4 = \mathbf{50} * F + 20 * I + 20 * U + 15 * D - 5 * V - 15 * O - 5 * T \tag{6.4}$$

$$f_5 = 40 * F + 20 * I + \mathbf{30} * U + 15 * D - 5 * V - 15 * O - 5 * T \tag{6.5}$$

f_1 represents the baseline fitness function used by the ESM. We proceed to modify the effects of DOM events (f_2), text input into fields (f_3), form submissions (f_4), and file uploads (f_5). This alteration aims to assess if adjustments to these parameters lead to notable performance changes in EvoCrawl.

Figure 6.2 illustrates EvoCrawl's performances under different fitness functions. EvoC_1 displays

	Insertions-EvoC	Insertions-BW	Insertions-AZ
WordPress	343	138	10
HotCRP	706	113	229
Humhub	667	18	0
Drupal	661	786	162
Kanboard	425	66	0
phpBB	93	116	25
ImpressCMS	1683	143	180
Opencart	671	0	62
Dokuwiki	68	38	0
Gitlab	1469	32	27

Table 6.2: Number of Insertions

	Forms-EvoC	Forms-BW	Froms-AZ
WordPress	12	8	3
HotCRP	44	6	8
Humhub	137	3	0
Drupal	124	142	9
Kanboard	76	5	0
phpBB	20	19	5
ImpressCMS	63	52	56
Opencart	20	0	7
Dokuwiki	16	9	0
Gitlab	42	7	1

Table 6.3: Number of Submitted HTML Forms

ESM results employing f_1 (Equation 6.1), while EvoC_2 reflects ESM outcomes with f_2 (Equation 6.2), and so forth. For most applications, parameter adjustments do not yield significant fluctuations in overall performance. In the case of WordPress, distinct fitness functions may display varying speeds in expanding coverage, yet they eventually converge to nearly the same code coverage. Conversely, Gitlab exhibits higher sensitivity to parameter changes, resulting in comparatively greater variability in code coverage outcomes.

6.3 Input Insertions

To detect a successful form submission, we search for the texts that scanners injected into each form. We log all the transactions that modify the tables in databases. If the texts injected by the scanners appear at any of the transactions, we consider the related form to be successfully submitted. We do not collect submissions other than HTML forms because they are hard to track. Also, this method cannot collect forms with the `GET` method since submitting these forms cannot modify the database. However, this experiment is mainly evaluating the ability of scanners on changing the states of applications so omitting forms with the `GET` method is not going to jeopardize the overall results. As BlackWidow and AuthZee generally exhibited the best performance out of the other crawlers, we compare the number of successful insertions and form submissions made by EvoCrawl, AuthZee, and BlackWidow in Table 6.2 and Table 6.3 respectively.

EvoCrawl successfully inserts more data and submits a greater number of forms compared to AuthZee in particular for Humhub, Dokuwiki, and Kanboard. To find specific forms in these three applications, a series of DOM events need to be triggered. Furthermore, AuthZee's fitness function does not capture and reward these events, consequently preventing it from finding and submitting these forms. However, AuthZee manages to submit nearly the same number of forms as EvoCrawl in applications like ImpressCMS. This can be attributed to the fact that the forms in ImpressCMS are straightforward and don't necessitate intricate interactions for submission. Despite AuthZee's evolutionary crawler not being specifically designed to find form-submission sequences, it can still manage to submit these relatively uncomplicated forms due to their simplicity.

Moreover, even when AuthZee manages to submit almost the same quantity of forms, EvoCrawl still outperforms significantly by inserting nearly ten times the amount of inputs into the Impress-CMS database. This achievement comes from EvoCrawl's Evolutionary Search Module (ESM), which specifically rewards sequences accomplishing successful input insertion within the application. This extends to the children or mutations of these successful sequences, as they are highly likely to continue the trend of data insertion. This behavior is strategically advantageous since even for identical forms, the filled contents can vary, effectively navigating the application into diverse states.

In the cases of phpBB, Dokuwiki, and ImpressCMS, BlackWidow manages to achieve only a marginally lower number of input insertions and HTML form submissions. This outcome can be attributed to the observation that numerous forms within these applications do not necessitate intricate logic for submission. As a result, the crawler does not need to try various sequences to accomplish form submission. However, BlackWidow outperforms EvoCrawl in the case of Drupal. This can be attributed to ESM's prolonged engagement with each page, resulting in a failure to locate all the forms within the 24-hour timeframe. Furthermore, CM encounters challenges in submitting specific forms due to its inability to infer the dependencies among DOM events. In Drupal's case, these form submissions necessitate the activation of multiple DOM events. As a result of these limitations, BlackWidow achieves more favorable results than EvoCrawl when tested on Drupal.

For HotCRP, Humhub, Kanboard, and Gitlab, EvoCrawl has a similar reason that makes it outperform BlackWidow. BlackWidow fails to submit certain forms because it either interacts with elements in the wrong order or enters the wrong values to certain input fields. Moreover, it further misses the forms that depend on the successful submission of these forms. For example, BlackWidow fails to submit a paper on Hotcrp and therefore cannot detect the forms that assign the paper and review the paper. It also failed to create tasks inside the project page on Kanboard, and cannot create new spaces on Humhub as well. In the case of Opencart, the submit buttons for all forms are not directly situated within the HTML form. Instead, they are associated with JavaScript functions that trigger the submission. However, as BlackWidow solely interacts with elements located inside the form for its form submission process, it inserts no inputs into the database.

6.4 IDOR Vulnerability Detectors

The public filter of EvoCrawl's IVD requires an attacker to replay the interactions of both the crawler module and the ESM to classify the collected URLs. We set the privilege level of the attacker to be the second highest and the crawler user to be the highest. AuthZee's public filter also requires a second user, whose privilege level is set to be the second highest as well.

	URLs	Private URLs	Public URLs	FP-1	FP-2	Vul.
WordPress	1025	379	646	9	106	0
HotCRP	526	415	111	39	3	0
Humhub	10729	9451	1278	0	5	0
Drupal	1908	1242	666	4	55	0
Kanboard	7973	4511	3462	17	0	0
phpBB	1684	1527	158	385	0	0
Opencart	1202	870	332	4	60	0
Dokuwiki	3121	864	2257	8	13	0
ImpressCMS	615	593	22	0	111	2
Gitlab	1382	640	742	27	63	1

Table 6.4: Results of EvoCrawl's IVD

	URLs	Private URLs	Public URLs	FP-1	FP-2	FP-3	Vul.
WordPress	525	311	214	131	3	27	0
HotCRP	405	378	29	55	0	7	0
Humhub	11503	11470	33	8746	0	22	0
Drupal	838	736	102	20	7	45	0
Kanboard	6	4	2	2	0	0	0
phpBB	659	281	378	216	0	0	0
Opencart	1958	1543	415	341	10	14	0
Dokuwiki	231	129	102	34	1	7	0
ImpressCMS	143	86	57	35	0	3	1
Gitlab	477	321	156	50	16	18	1

Table 6.5: Results of AuthZee's VD

Table 6.4 and Table 6.5 displays the results of the EvoCrawl's IVD and AuthZee's VD. We collect various metrics to assess their performance. These include the total number of URLs discovered, the number of URLs classified as private, and the number of URLs classified as public. False positives are further categorized into three types. The first type denoted as FP-1, occurs when resources are missed by the public filter and thus incorrectly classified as private. The second type referred to as FP-2 arises when resources are correctly classified as private but do not contain any sensitive information. These resources typically include JavaScript files or non-sensitive media such as logos or emojis. The third type represented as FP-3 happens when the similarity ratio threshold is not appropriately suited for specific applications, leading to inaccurate identification as vulnerabilities.

The count of FP-1 instances generated by AuthZee's VD considerably surpasses that in EvoCrawl's IVD. This discrepancy primarily stems from AuthZee's dual-user crawling on different sections of applications. Moreover, in scenarios like Humhub, FP-1 instances also originate from URL dynamism. Specifically, Humhub assigns timestamps to URLs related to certain resources. Consequently, when two users access the same page, distinct URLs are recorded due to timestamp variations. This subsequently leads to misclassification and the occurrence of FP-1 instances.

In the case of EvoCrawl, the IVD relies on a sitemap to classify resources as public or private. However, misclassifications can occur when certain paths are not included in the sitemap, leading to FP-1. This issue is particularly evident in applications like HotCRP and phpBB, where a significant number of URLs are misclassified as private. Upon closer inspection, it was found that some of these URLs, although labeled as private, are actually inaccessible through the user interface (UI). Therefore, the misclassification is not solely due to missing paths in the sitemap but rather results from non-existent paths that attackers cannot navigate. The sitemap is continually updated during scanning by CM and ESM. Nonetheless, time constraints may prevent the discovery of all possible paths, resulting in misclassifications.

As for the FP-2, Both AuthZee's VD and EvoCrawl's IVD are unable to determine the sensitivity of the information disclosed by private resources. Each application's developers have their own criteria for determining what constitutes sensitive data. While it is ideal to prevent the disclosure of any private information to attackers, it can be challenging for developers to enforce protection measures on all private resources, particularly those that do not contain sensitive data. Therefore, the IVD focuses on identifying IDOR vulnerabilities without making judgments on the sensitivity of the disclosed information.

In contrast to AuthZee, where EvoCrawl's IVD mainly employs keyword matching to identify correct protection of private resources, thus eliminating the generation of FP-3 instances. Conversely, AuthZee's VD employs a similarity ratio between responses from different users to check resource access control. However, identifying a universal similarity threshold suitable for every application is challenging, resulting in type 3 false positives.

Overall, the IVD discovers 3 zero-day IDOR vulnerabilities including two that have been detected by AuthZee. We report the new vulnerability to the ImpressCMS developers but has not received a response yet.

6.5 XSS Vulnerability Detectors

Despite the similarity between EvoCrawl's XSS Vulnerability Detector and BlackWidow, we conduct an evaluation to assess its performance, considering the crucial role of the crawler in XVD's effectiveness. The evaluation of EvoCrawl reveals five vulnerabilities and one injection point in different web applications.

In Humhub, EvoCrawl discovers one injection point that allows the website owner to inject a custom script for tracking page statistics. Although this is not considered a bug, EvoCrawl successfully detects the injection point.

For WordPress,EvoCrawl identifies two stored XSS vulnerabilities, which can only be exploited by admin or editor users. The developers have decided not to address these vulnerabilities according to their security policy.

In HotCRP, we find one stored XSS vulnerability and one reflected XSS vulnerability. The stored XSS vulnerability has been acknowledged by the developers and will be fixed in future versions of the applications. The reflected XSS bug in HotCRP, however, cannot be exploited by attackers as it is only visible to admin users and protected by a CSRF token.

For Kanboard, EvoCrawl successfully identifies one stored XSS vulnerability, which has been reported to the developers and will be patched in future versions.

6.6 Vulnerabilities Found

In total, EvoCrawl has identified 6 zero-day vulnerabilities in popular web applications such as WordPress, HotCRP, Kanboard, and ImpressCMS. Out of these, 4 vulnerabilities have been acknowledged and confirmed by the developers. The details of each vulnerability are as follows:

- WordPress (acknowledged but not fixed): The two XSS injection points of WordPress are the comment field and the post title field. Both of these fields lack proper sanitization, allowing editor users or admin users to inject custom scripts into them. According to the WordPress security policy, XSS injection points that can only be exploited by higher-level users will not be fixed. To detect the "post title field" vulnerability, the crawler initially needs to generate a post, setting the title value as the XSS payload. This action steers the application into a new state, wherein a new post is formed and saved. Only within this state can the crawler navigate to the new post page, facilitating the identification of XSS payload execution.

- HotCRP [11] (acknowledged and fixed): One XSS injection point on `settings/decisions` page. Chair or admin users can inject custom scripts into the decision name field. To exploit the identified vulnerability, the crawler must explore various states of the application. Initially, it navigates the application to a specific state s_1, where the decision name is altered to incorporate the XSS payload. Subsequently, the application needs to be driven to another state s_2 by submitting a paper. Finally, the XSS payload's execution is viable on the review page of the submitted paper.

- HotCRP (reported but not acknowledged): EvoCrawl identified a reflected XSS injection point on the `settings/reviews` page, specifically in the round name field. However, this injection

point is only accessible to admin users and is protected by a CSRF token, so the developer does not consider it a vulnerability.

- Kanboard (acknowledged and fixed) [13]: One XSS injection point on `settings/application` page, enabling admin users to inject scripts to the application URL field. To expose the vulnerability, the crawler follows a specific procedure. Initially, it modifies the application URL field to include the XSS payload. Then, the crawler navigates to the `settings/api` page to initiate the execution of the payload.
- ImpressCMS (reported but not responded): An IDOR vulnerability on endpoint `/libraries/image-editor/image-edit.php?image_id=1&uniq=`. Attackers can force browsing to the private images by changing the `image_id`. To identify the vulnerability, the crawler's process entails initially uploading an image and subsequently locating the IDOR (Insecure Direct Object References) endpoint.

Chapter 7

Limitations

Parameter Tuning. For optimal results, EvoCrawl currently requires manual parameter adjustments within the fitness function. In the future, we aim to conduct a more comprehensive analysis of the influence of each parameter. Our goal is to design a system that can autonomously fine-tune these parameters, enabling adaptation to the specific needs of each tested application.

Manual Configuration. To ensure the stability of the crawling process, we implement measures to prevent EvoCrawl from crawling on user pages, configuration pages, and plugin installation pages. Given the diverse URLs associated with these pages across different applications, users are required to manually gather the relevant keywords and supply them to the crawler. Additionally, in an effort to enhance efficiency, we always keep the crawler in the logged-in state by circumventing interactions with the logout button.

Seed Selection. In many cases, various URLs can direct to the same or highly similar pages within applications. For example, pages displaying objects with different sorting criteria may possess distinct URLs. As EvoCrawl relies on page URLs as seeds for crawling, there's a potential for redundancy, where the tool might spend time crawling pages it has already processed, thereby decreasing efficiency further. In future developments, we will try to implement more effective methods for distinguishing between different pages. Instead of relying solely on page URLs, we aim to employ techniques such as DOM comparison to achieve greater accuracy and precision.

Chapter 8

Related Work

8.1 White-box Scanners

[17, 18, 30, 29] detect access-control vulnerabilities in a white-box manner.

Mace [17] generates a 4-tuple object for each user role to represent the associated access-control level by program analysis. By matching the access-control level with the conditional statements for each sink (query), mace can identify the sinks inconsistent with the authorization context and treat them as privilege escalations.

Daniel [18] try to find missing security checks by using a catalog of access control patterns. These broken access control patterns are generated by doing large-scale case studies on the existing found vulnerabilities. If any checks in the tested program match with one of the patterns, it will be considered a possible vulnerability.

Fix me up [29] can automatically detect and repair access control vulnerabilities but they require manual annotations on the source code.

Similar to EvoCrawl, Fangqi, et al. [30] also build a site map for each user role. They filter out the public pages by comparing the site maps of different users and force browsing to the private pages to check access control. However, only doing state analysis can lead to missing certain links, since some of them are generated during the running time. Overall, using white-box methods is always limited by the type of source codes, therefore, hard to be generalized for all websites or web applications.

8.2 Black-box Scanners

BlackWidow by Benjamin et al. [9] and jAk by Giancarlo et al. [24] also build their navigation graph with client-side JavaScript events. However, they additionally capture the events having no mutations on the current DOMs, causing the crawlers to waste extra time exploring these events and decrease the overall performance. Furthermore, they capture the JS events by hacking the `addEventListener` function dynamically each time a page has been loaded. This method is not robust enough and sometimes misses some of the events on the page.

Enemy of the State by Adam et al. [6] uses static links and forms to build the navigation graph but misses the JavaScript events on the client side. Since many modern web applications heavily

rely on SPA (single-page applications) and AJAX techniques, it is hard for Enemy of the State to fully explore these websites.

Crawljax by Ali et al. [16] uses a state machine to guide the crawling process. Each state represents a unique DOM (Document Object Model) of the web page. However, CrawlJAX cannot track the dependencies among different states.

There are other black-box scanners [23, 22, 8] but they mainly focus on the vulnerability detection part. LigRE [8] is able to reverse-engineer the web application as a control- and data-flow model and use them to guide the fuzzing for the detection of XSS vulnerabilities. Deemon [23] is able to detect the CSRF vulnerability by modeling the behavior of the application, while Giancarlo and Davide [22] detect logic errors with analysis on the interaction traces.

8.3 Grey-Box Scanners

In recent times, the grey-box fuzzing technique has gained traction for testing web applications [26, 10, 32]. Inspired by AFL, webFuzz [26] applies instrumentation to PHP applications at the Abstract Syntax level to achieve branch coverage. It integrates a dynamic (Black-box) crawler to generate requests, which serve as new seeds for the fuzzing process. Moreover, webFuzz employs an attack grammar to mutate these seeds and effectively increase the coverage. BackREST [10] starts by crawling the web application and collecting the REST API. Then it utilizes the gathered API to generate and mutate inputs for the fuzzer. Additionally, BackREST incorporates taint analysis to track which inputs reach the security-sensitive locations. However, it is worth noting that [26] is specifically designed for testing PHP applications and [10] only supports Node.js, limiting their applicability. While Witcher [32] supports testing on multiple languages, its effectiveness is still limited by the performance of the black-box scanner it incorporates.

8.4 Evolutionary Search

Using evolutionary search to crawl web applications was inspired by AuthZee [14]. AuthZee uses evolutionary search to capture different requests and page URLs but misses exploring different states of the application.

KameleonFuzz designed by Fabian et al. [7] uses a genetic algorithm to generate payloads that are more likely to pass through the server sanitizer and find an XSS vulnerability. Marashdih et al. [15] use static analysis to generate Control Clow Graphs (CFGs) that contain potentially vulnerable paths. Then they employ the genetic algorithms to generate XSS scripts and use them as inputs to verify the feasibility of these vulnerable paths. Similarly, Moataz A. and Fakhreldin [1] utilize third-party tools to generate potentially vulnerable paths and then employ genetic algorithms to verify their feasibility. However, the performance of their approach is constrained by the capabilities of the third-party tool, as it may not be able to find all potentially vulnerable paths. The genetic algorithm employed by these web works is used to generate fuzz inputs but not for exploring the application codes and states. On the other hand, EvoCrawl's goal is to try revealing as many sources as possible but not generate inputs that successfully pass the sanitizer check.

Chapter 9

Conclusion

In conclusion, we find that being able to explore both server- and client-side web application state enables a standard crawler to reach more code. By combining standard crawling in its Crawler Module with an Evolutionary Search Module that focuses on exploring application state, EvoCrawl is able to obtain greater code coverage and submit more forms than previous state-of-the-art web application scanners. Using EvoCrawl, we have also found 8 new vulnerabilities in popular and heavily used web applications.

Bibliography

[1] Moataz A. Ahmed and Fakhreldin Ali. "Multiple-path testing for cross site scripting using genetic algorithms". In: *Journal of Systems Architecture* 64 (2016). Real-Time Signal Processing in Embedded Systems, pp. 50–62. ISSN: 1383-7621. DOI: https://doi.org/10.1016/j.sysarc.2015.11.001. URL: https://www.sciencedirect.com/science/article/pii/S1383762115001332.

[2] *Burp Suite - Application Security Testing Software - PortSwigger*. 2022. URL: https://portswigger.net/burp.

[3] *Coverband*. 2023. URL: https://github.com/danmayer/coverband.

[4] Data and Vulnerability Management Analysis. *The Rise of IDOR — HackerOne*. 2021. URL: https://www.hackerone.com/data-and-analysis/rise-idor.

[5] Adam Doupé, Marco Cova, and Giovanni Vigna. "Why Johnny Can't Pentest: An Analysis of Black-Box Web Vulnerability Scanners". In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Ed. by Christian Kreibich and Marko Jahnke. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 111–131. ISBN: 978-3-642-14215-4.

[6] Adam Doupé et al. "Enemy of the State: A State-Aware Black-Box Web Vulnerability Scanner". In: *21st USENIX Security Symposium (USENIX Security 12)*. Bellevue, WA: USENIX Association, Aug. 2012, pp. 523–538. ISBN: 978-931971-95-9. URL: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/doupe.

[7] Fabien Duchene et al. "KameleonFuzz: Evolutionary Fuzzing for Black-Box XSS Detection". In: *Proceedings of the 4th ACM Conference on Data and Application Security and Privacy*. CODASPY '14. San Antonio, Texas, USA: Association for Computing Machinery, 2014, pp. 37–48. ISBN: 9781450322782. DOI: 10.1145/2557547.2557550. URL: https://doi.org/10.1145/2557547.2557550.

[8] Fabien Duchène et al. "LigRE: Reverse-Engineering of Control and Data Flow Models for Black-Box XSS Detection". In: *2013 20th Working Conference on Reverse Engineering (WCRE)*. 2013, pp. 252–261. DOI: 10.1109/WCRE.2013.6671300.

[9] Benjamin Eriksson, Giancarlo Pellegrino, and Andrei Sabelfeld. "Black Widow: Blackbox Data-driven Web Scanning". In: *2021 IEEE Symposium on Security and Privacy (SP)*. 2021, pp. 1125–1142. DOI: 10.1109/SP40001.2021.00022.

[10] François Gauthier et al. "BackREST: A Model-Based Feedback-Driven Greybox Fuzzer for Web Applications". In: *ArXiv* abs/2108.08455 (2021).

[11] *HotCRP XSS*. 2023. URL: https://github.com/kohler/hotcrp/commit/d4ffdb0ef806453c54ddca7fdda3e5c60356285c.

[12] *Insecure Direct Object References*. 2013. URL: https://wiki.owasp.org/index.php/Top_10_2013-A4-Insecure_Direct_Object_References.

[13] *Kanboard XSS*. 2023. URL: https://github.com/kanboard/kanboard/pull/5119.

[14] Akshay Kawlay. "Evolutionary Search for Authorization Vulnerabilities in Web Applications". MA thesis. University of Toronto (Canada), 2021.

[15] Abdalla Wasef Marashdih, Zarul Fitri Zaaba, and Herman Khalid Omer. "Web Security: Detection of Cross-Site Scripting in PHP Web Application using Genetic Algorithm". In: *International journal of advanced computer science and applications* 8.5 (2017).

[16] Ali Mesbah, Engin Bozdag, and Arie van Deursen. "Crawling AJAX by Inferring User Interface State Changes". In: *2008 Eighth International Conference on Web Engineering*. 2008, pp. 122–134. DOI: 10.1109/ICWE.2008.24.

[17] Maliheh Monshizadeh, Prasad Naldurg, and V. N. Venkatakrishnan. "MACE: Detecting Privilege Escalation Vulnerabilities in Web Applications". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. CCS '14. Scottsdale, Arizona, USA: Association for Computing Machinery, 2014, pp. 690–701. ISBN: 9781450329576. DOI: 10.1145/2660267.2660337. URL: https://doi.org/10.1145/2660267.2660337.

[18] Joseph P. Near and Daniel Jackson. "Finding Security Bugs in Web Applications Using a Catalog of Access Control Patterns". In: *Proceedings of the 38th International Conference on Software Engineering*. ICSE '16. Austin, Texas: Association for Computing Machinery, 2016, pp. 947–958. ISBN: 9781450339001. DOI: 10.1145/2884781.2884836. URL: https://doi.org/10.1145/2884781.2884836.

[19] *OWASP Top Ten*. 2017. URL: https://owasp.org/www-project-top-ten/2017/Top_10.

[20] *OWASP Top Ten*. 2021. URL: https://owasp.org/www-project-top-ten/.

[21] *OWASP Zed Attack Proxy (ZAP)*. 2023. URL: https://www.zaproxy.org/.

[22] Giancarlo Pellegrino and Davide Balzarotti. "Toward Black-Box Detection of Logic Flaws in Web Applications". In: *NDSS 2014, Network and Distributed System Security Symposium, 23-26 February 2014, San Diego, USA*. Ed. by ISOC. ISOC. Personal use of this material is permitted. The definitive version of this paper was published in NDSS 2014, Network and Distributed System Security Symposium, 23-26 February 2014, San Diego, USA and is available at : http://dx.doi.org/10.14722/ndss.2014.23021. San Diego, 2014.

[23] Giancarlo Pellegrino et al. "Deemon: Detecting CSRF with Dynamic Analysis and Property Graphs". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 1757–1771. ISBN: 9781450349468. DOI: 10.1145/3133956.3133959. URL: https://doi.org/10.1145/3133956.3133959.

[24] Giancarlo Pellegrino et al. "jÄk: Using Dynamic Analysis to Crawl and Test Modern Web Applications". In: *Research in Attacks, Intrusions, and Defenses*. Ed. by Herbert Bos, Fabian Monrose, and Gregory Blanc. Cham: Springer International Publishing, 2015, pp. 295–316. ISBN: 978-3-319-26362-5.

[25] *php-code-coverage*. 2023. URL: https://github.com/sebastianbergmann/php-code-coverage.

[26] Orpheas van Rooij et al. "WebFuzz: Grey-Box Fuzzing for Web Applications". In: *Computer Security – ESORICS 2021: 26th European Symposium on Research in Computer Security, Darmstadt, Germany, October 4–8, 2021, Proceedings, Part I*. Darmstadt, Germany: Springer-Verlag, 2021, pp. 152–172. ISBN: 978-3-030-88417-8. DOI: 10.1007/978-3-030-88418-5_8. URL: https://doi.org/10.1007/978-3-030-88418-5_8.

[27] *rrweb: record and replay the web*. 2018. URL: https://github.com/rrweb-io/rrweb.

[28] *Skipfish - Web Application Security Scanner*. 2022. URL: https://code.google.com/archive/p/skipfish/.

[29] Sooel Son, Kathryn S. McKinley, and Vitaly Shmatikov. "Fix Me Up: Repairing Access-Control Bugs in Web Applications". In: *Network and Distributed System Security Symposium*. 2013.

[30] Fangqi Sun, Liang Xu, and Zhendong Su. "Static Detection of Access Control Vulnerabilities in Web Applications." In: *USENIX Security Symposium*. Vol. 64. 2011.

[31] *TestCafe*. 2023. URL: https://testcafe.io/.

[32] Erik Trickel et al. "Toss a Fault to Your Witcher: Applying Grey-box Coverage-Guided Mutational Fuzzing to Detect SQL and Command Injection Vulnerabilities". In: *Proceedings of the IEEE Symposium on Security and Privacy*. May 2023.

[33] *w3af*. 2022. URL: http://w3af.org/.

[34] *Xdebug-Code Coverage Analysis*. 2023. URL: https://xdebug.org/docs/code_coverage.

[35] xss. *Cross-site Scripting (XSS)*. 2018. URL: https://wiki.owasp.org/index.php/Cross-site_Scripting_(XSS).

[36] Jun Zhu et al. "Mitigating Access Control Vulnerabilities through Interactive Static Analysis". In: *Proceedings of the 20th ACM Symposium on Access Control Models and Technologies*. SACMAT '15. Vienna, Austria: Association for Computing Machinery, 2015, pp. 199–209. ISBN: 9781450335560. DOI: 10.1145/2752952.2752976. URL: https://doi.org/10.1145/2752952.2752976.